LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT
School of Energy Systems
Electrical Engineering

Timo Saxell

LIGHTWEIGHT SCHEDULER FOR REAL TIME EMBEDDED SYSTEM FIRMWARE DESIGN

Examiners: Professor Pertti Silventoinen
D.Sc. Janne Hannonen

Abstract

Lappeenranta University of Technology

School of Energy Systems

Electrical Engineering

Timo Saxell

Lightweight scheduler for real time embedded system firmware design

Masters thesis 2019

45 pages, 8 figures, 3 appendices

Examiners: Professor Pertti Silventoinen

D.Sc. Janne Hannonen

Supervisor: D.Sc. Janne Hannonen

Keywords: real-time operating system, scheduler

The firmware development of Efore Powernet Oy needs to have a functioning way to improve the design time and maintainability in embedded system design projects. One way of improvement is to introduce a real-time operating system to the development. This thesis introduces the requirements for a real-time operating system from the viewpoint of Efore Powernet Oy's firmware development and assesses the feasibility of the real-time operating systems available.

In addition to the review, this thesis describes the design process of a custom scheduler and also includes a study on the impact of the custom scheduler to the Efore Powernet Oy's firmware development. The study shows improvement in the firmware performance, the maintainability and design time in the embedded design process.

Tiivistelmä

Lappeenrannan Teknillinen Yliopisto

School of Energy Systems

Electrical Engineering

Timo Saxell

Kevyt aikatauluttaja reaaliaikaisten sulautettujen järjestelmien laitteisto-ohjelmien suunnitteluun

Diplomityö 2019

45 sivua, 8 kuvaa, 3 liitettä

Työn tarkastajat:	Professori Pertti Silventoinen
	TkT Janne Hannonen
Työn ohjaaja:	TkT Janne Hannonen
Hakusanat:	reaaliaikakäyttöjärjestelmä, aikatauluttaja

Efore Powernet Oy:n laitteisto-ohjelmistotuotekehitys tarvitsee keinoja sulautettujen järjestelmien suunnitteluprojektien suunnitteluajan supistamiseen ja ylläpidettävyyden kehittämiseen. Yksi parannustapa on reaaliaikakäyttöjärjestelmän käyttöönottaminen.Tämä diplomityö esittelee Efore Powernet Oy:n laitteisto-ohjelmistotuotekehityksen reaaliaikakäyttöjärjestelmän vaatimukset ja tutkii, löytyykö valmista käyttöjärjestelmää, joka täyttää nuo vaatimukset.

Lisäksi tämä diplomityö esittelee käyttöjärjestelmän aikatauluttajan suunnitteluprosessin ja tutkii myös kehitetyn aikatauluttajan vaikutusta Efore Powernet Oy:n laitteisto-ohjelmistotuotekehitykseen. Tutkimus osoittaa parannuksia sulautettujen laitteisto-ohjelmistojen suorituskyvyssä, ylläpidettävyydessä ja suunnitteluajassa.

Acknowledgements

This study was carried out at Efore Powernet Oy. I would like to thank my colleague Mihail Velichkov for helping me with the ideas and concepts for the design process and constant sparring to get the scheduler done.

Very special thanks go to D.Sc. Janne Hannonen for helping me big time to get this design process into an actual academic paper form.

I want to also thank the people at LUT for their quick support on bureaucratic parts and especially Professor Pertti Silventoinen for his continued support.

Biggest thanks goes to my wife Heta for her never-ending support, help and belief during the whole process.

Contents

Abbreviations and symbols

t	time

ADC	Analog to Digital Converter
CAN	Controller Area Network
DAC	Digital to Analog Converter
FRC	Free Running Counter
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
RAM	Random Access Memory
RTOS	Real Time Operating System
SCU	System Controller Unit

1 Introduction

Microcontrollers and further embedded systems can be stated to be the heart and soul of modern electronics. Since the introduction of affordable microcontrollers in 1970's, they have been used in multiple applications, including controlling power electronics in power converters.

As time has passed, the requirements for the power converters have become more complex due to the need of increased efficiency, self diagnostics, communication interfaces and many more features. This added complexity makes designing the device firmware for power converters increasingly more difficult and time consuming.

The demand for short time to market, or in other words, time from an idea to a prototype and even further into a commercial product in the modern, highly competed power supply markets is often controversial with the time consuming designs. This has been seen as a problem also at Efore Powernet Oy.

Efore Powernet Oy is a Finnish company designing and selling a wide range of power conversion products for industrial and rail customers. Many of the products are designed for customers specific needs.

This thesis studies how the embedded firmware development of Efore Powernet Oy can be improved by introducing a real-time operating system into the firmware design process.

1.1 Embedded systems

An embedded system is considered to be any electronic device which has an integrated microcontroller. A microcontroller is a microprocessor with integrated peripherals and memory. The embedded devices are considered to have less powerful controllers. Therefore computers with usually more powerful processors are not included (LaVerne 1989; Heath 2003). Embedded systems have the application firmware and possible operating system bundled together, but the operating system is not obligatory.

The purpose of the embedded system is usually very specific, for example a smart battery charger. The charger could have a very standard power supply design where the microcontroller adds the charging function by i.e. controlling the voltage and current of the power supply to fit the specific charging curves for the battery load (Mundra & Kumar 2007).

A microcontroller is a system on a chip. It contains one or more processing cores, random access memory (RAM), usually some non-volatile memory for the firmware, and variable amount of peripherals for measuring, controlling or interaction with the surroundings of the embedded device. For example analog to digital converters (ADC), which measure physical quantities that are converted in to voltage levels, digital to analog converters (DAC) can be used to drive voltage controlled actuators. Communication buses allow the microcontroller to interact with other microcontrollers or computer systems.

Designing application firmware for embedded system requires low level knowledge of the hardware capabilities of the microcontroller. The designer needs to know which peripherals (timers, analog to digital converters, digital to analog converters, communication buses etc.) are needed for the application and how they need to be initialized.

1.2 Real time operating systems

Traditional way of designing firmware for embedded systems is to employ a looping structure as the main part of the application and to use interrupts to handle time critical events. This approach is called foreground / background systems, where the interrupt handlers are the foreground and the main loop is the background or so called round-robin. When the code complexity increases, this approach becomes more difficult to handle (Kalman 2010, pp. 11–12). Fig 1. illustrates the functionality of this approach.

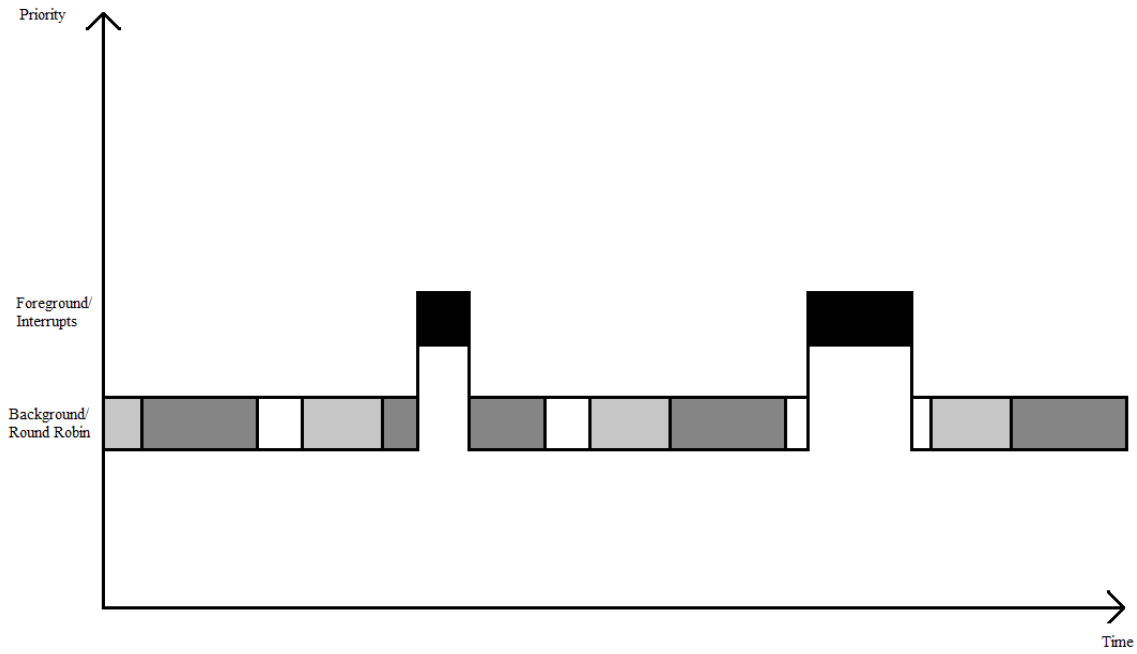

Fig 1. Illustration of a foreground / background (round-robin) scheduled system. The background functions are executed in a fixed order and their execution is interrupted by foreground functions launched by hardware or software interrupt service routines.

A real time operating system (RTOS) differs from the round-robin by dividing the design into smaller parts called tasks, which are given priorities. The highest priority tasks are executed first and the lowest priority tasks are executed last. RTOSes are exploited for helping the design of time critical embedded system applications by automating switching between different tasks required from the application. RTOS also helps the management of the resources of the microcontroller (Labrosse 2011, pp. 31–35).

RTOS differs from computer operating systems as the purpose of the latter is to provide the user with the possibility to run and install different kinds of applications. RTOS provides tools and services for the firmware designer, not the end user specifically.

1.3 Contents of real time operating systems

The tools and services an RTOS offers varies between RTOSes. All RTOSes have a timing algorithm to schedule the tasks of the firmware. Also RTOSes may contain messaging services to make the intertask data exchange easier.

Usually RTOSes also contain ways to protect resources and critical parts of the code (Nemati 2012). These are called mutex (mutual exclusion) and semaphor. A mutex blocks access to certain variable(s) or area of memory and a semaphor can be reserved to block other tasks calling the same critical part of the code. After the task is finished with the resource or code section, then the mutex or semaphor is released to allow other tasks to access the shared resource (Labrosse 2011, pp. 35–39).

1.4 Scheduler

Scheduler is the part of the RTOS which is responsible for choosing which task to run next. There are two primary approaches to scheduling: A pre-emptive scheduler and a co-operative scheduler (non pre-emptive).

A pre-emptive scheduler is able to switch tasks immediately when a higher priority task becomes ready to run. This means that the lower priority task will be suspended, when the scheduler changes the task. When the higher priority task has finished, then the lower priority task can continue to run. Fig 2. illustrates the functionality of pre-emptive scheduling.

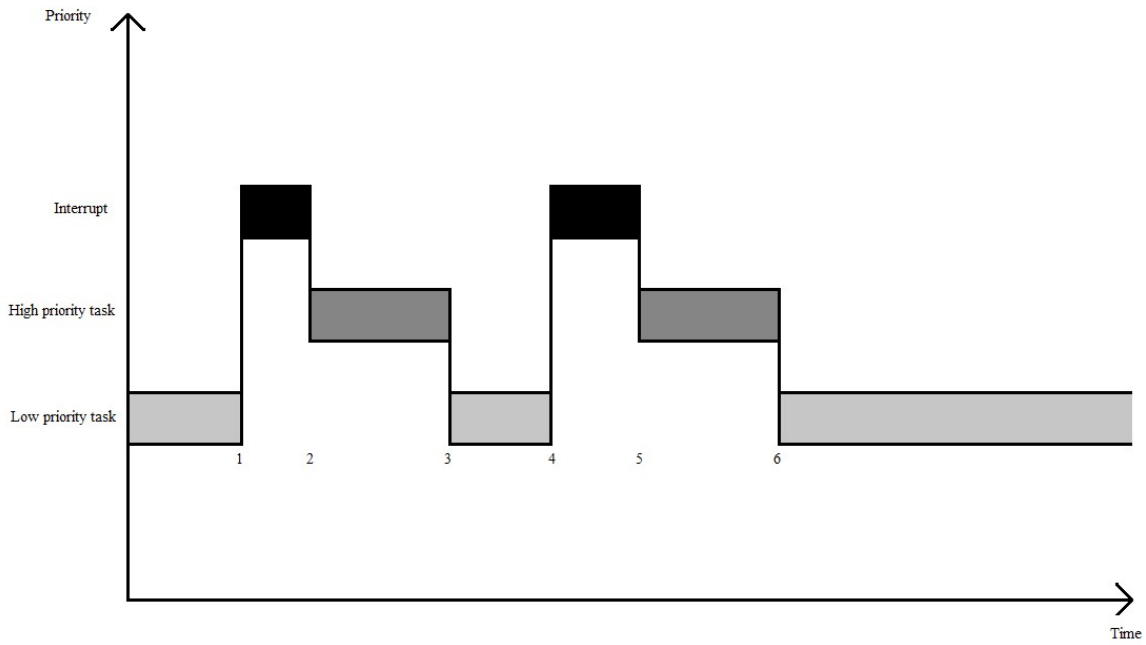

Fig 2. Illustration of a pre-emptively scheduled system. At t=1 and t=4 a low priority task gets interrupted by an interrupt. At t=2 and t=5 the interrupt enables a high priority task to run and at t=3 and t=6 the high priority task finishes and waits for the next enabling event and the execution switches back to the low priority task.

A co-operative scheduler does not suspend the lower priority task. After the lower priority task has finished, the scheduler selects the higher priority task to run. Fig 3. illustrates the functionality of co-operative scheduling.

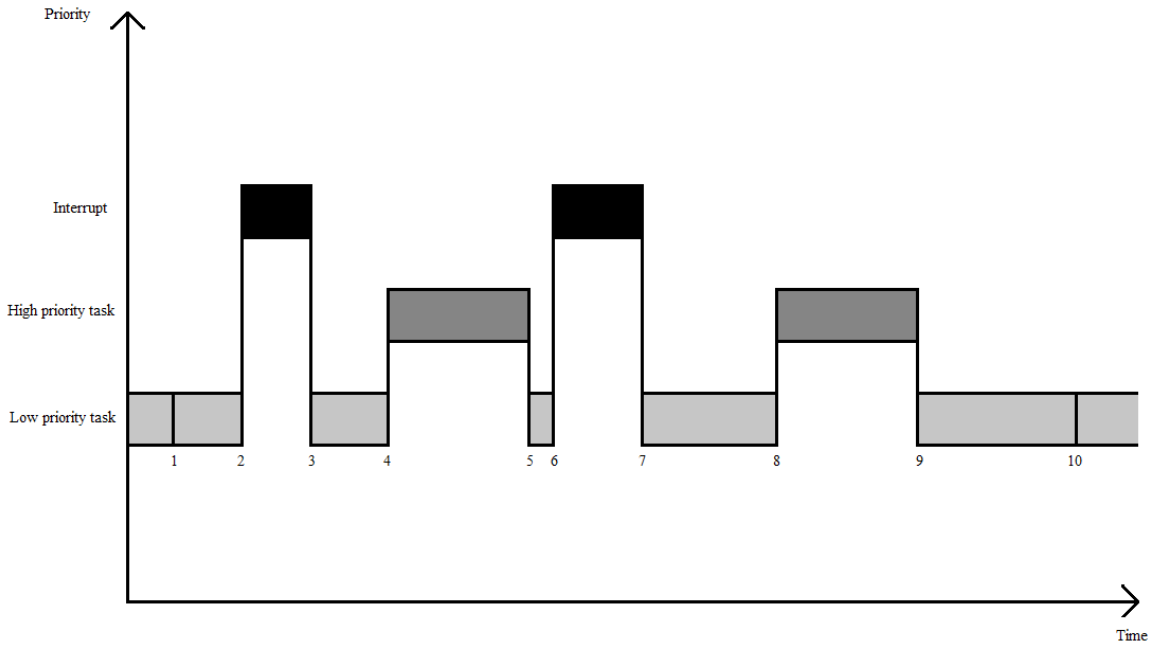

Fig 3. Illustration of a co-operatively scheduled system. At t=1 and t=10 the low priority task is finished and as it is the only non blocked task, it gets executed again. At t=2 and t=6 the low priority task gets interrupted and the high priority task gets unblocked and the execution returns at t=3 and t=7 to the low priority task. At t=4 and t=8 the low priority task is finished and as the high priority task is unblocked now, it gets executed. At t=5 and t=9 the high priority task finishes and waits for the next enabling event and the execution switches back to the low priority task.

A pre-emptive scheduler is more deterministic as the delay from an event causing the high priority task to become ready to the actual execution of the task is constant. On the other hand, the pre-emptiveness requires more overhead time from the processor to handle the context switching between the tasks and also the scheduler.

The signal for pre-emptive scheduler to run and check the available tasks is called a tick or a system tick. A shorter tick period makes the systems response more real time, but it also increases the system overhead, as the scheduler is called more often. A co-operative scheduler on the other hand requires more attention from the application firmware designer to reach required level of determinism. This can be reached by making lower priority tasks shorter than the required response time of higher priority tasks and also dividing the workload of the functionalities across multiple tasks.(Labrosse 2011, pp. 151–163; Kalman 2010, pp. 19–21).

1.5 Scope of the thesis

This thesis describes the design process of a custom scheduler including reviewing the need and the usage advantages for the custom design. The goal shall be reached by dividing the focus on three topics.

First topic is to review three existing real time operating systems and to go through their advantages and disadvantages from the Efore Powernet Oy's firmware development viewpoint. This gives the background for designing a custom scheduler for Efore Powernet Oy's needs. This research is done by reviewing the literature and documentation of the selected real time operating systems.

Second topic in the thesis is the actual design and implementation process of a non pre-emptive scheduler. The assessment of this topic is based on the performance quality of the achieved result of the design and implementation process. The implementation is done in C programming language.

Third topic studies the impact of the custom scheduler to the firmware development. Two project examples are reviewed from the viewpoint of using the custom scheduler. This is an experimental study on what are the advantages on the scheduler based design over the traditional round-robin based embedded system firmware development.

The above stated research topics are assessed using qualitative methods.

From these topics the following research questions are derived:

- Is there a real time operating system available that would suit Efore Powernet Oy's firmware design needs?
- What kind of scheduler can meet the requirements of Efore Powernet Oy's firmware development?
- What kind of impact a scheduler has on Efore Powernet Oy's firmware development?

2 Reviewing existing real time operating systems

There are a great number of RTOSes available in the market. The range covers operating systems for various specialized needs. Some of them are open source and others are commercial. The size, complexity, and required processing capabilities of the available operating systems varies a lot. The RTOSes, especially commercial ones have been most often ported for multiple microcontroller platforms

In this chapter a few different RTOSes are selected for a review. Their capabilities and features are compared against each other and against the needs of Efore Powernet Oy's firmware development.

2.1 Selecting RTOSes for comparison

Three aspects define the main focus on selecting the RTOSes for comparison. First criteria is the scheduler type, pre-emptive or co-operative. Second criteria is the size and complexity, large or small. Third criteria is the commercial status of the RTOS, is it free or proprietary.

2.1.1 Efore Powernet Oy's RTOS needs

The review is done in contrast to the needs of Efore Powernet Oy's firmware development. As Efore Powernet Oy's market focus is in custom design power supplies, the requirements for the device firmware vary based on the end customer needs. Some of the most critical usage environments considered here are the auxiliary power supply systems for trains. The systems charge the on-board batteries and act as a power source to the critical train control systems.

In order for a train to be able to operate, uninterrupted operation of the systems is

mandatory.

This level of criticality introduces strict requirements for an RTOS.

- All peripheral drivers and their usage must be thoroughly known by the firmware developers. When dealing with critical power supply systems, all interaction between the firmware and the surrounding hardware must be known down to the physical pin states of the microcontroller.

  → The RTOS must be device independent.

- There must not be dynamic memory allocation as that opens a possibility for memory leaks and fragmentation. All such possibilities must be avoided in critical power systems.

  → All memory allocation must be done during compilation.

- Nothing in the operating system can interfere with the execution of the interrupts. If the operating system requires the interrupts to be disabled for it's own operation, that could damage the power system or may in the extreme circumstances cause a risk/danger of life, if handling of a critical interrupt gets delayed.

  → Operating system must not block interrupts.

- The RTOS must be able to launch tasks with programmable delays. These delays are required for easier and faster implementation of periodic control loops and watchdog timeouts.

- The RTOS must also be easily bypassed so that it does not hinder proprietary firmware sections, that are not under RTOS control. This means that the operating system cannot be given total control over code execution as a part of the firmware code needs to be executed with minimum delay. This is required so that the most critical functions can be called safely as often as possible, if needed.

  → The tasks must be functions with clearly defined beginning and end. No infinite loops allowed. The tasks must perform their duty and be automatically blocked until called upon again.

2.2 µC/OS-III

First selection to the review is Micrium's µC/OS-III originally designed by Jean J. Labrosse. The first version of the µC/OS was released in 1991. It is one of the longest running commercial RTOSes. Micrium's combined market share in 2017 was 9%, µC/OS-II and µC/OS-III combined (AspenCore 2017).

µC/OS-III was chosen to represent a pre-emptive, considerably complex and proprietary RTOS.

2.2.1 Review against Efore Powernet Oy's needs

µC/OS-III does not define the peripheral drivers itself, but encourages the usage of the standard drivers from the microcontroller manufacturer (Labrosse 2011, pp. 50–51).

µC/OS-III acknowledges the dangers involved in dynamic memory management and prefers static creation of memory partitions which are controlled by the operating system. The operating system can be requested to dynamically allocate memory from the pre-created partitions thus removing most of the risks associated with dynamic memory allocation (Labrosse 2011, pp. 343–354). However, this does not take into consideration a flaw in the application design to assign too little space for the memory partitions. This can cause the RTOS to not be able to provide memory for new tasks, messages etc. and thus create a possible error situation that might not be able to be tested.

µC/OS-III allows firmware to have non-kernel-aware interrupt service routines (ISR). This means that the operating system doesn't interfere with the execution of how interrupts are handled (Labrosse 2011, pp. 175–195). The RTOS has critical sections, where interrupts are disabled. This introduces possible latency to interrupt handling.

µC/OS-III has multiple options to handle task timing. It can use the multiple of the operating system's timing ticks for delaying from milliseconds up to hours (Labrosse 2011, pp. 203–212).

µC/OS-III doesn't allow sections of the firmware to be outside of the control of the RTOS.

All of the functionality of the firmware must be divided into the tasks (Labrosse 2011, pp. 91–140).

2.2.2 Conclusion

Memory management does not fulfil the needs completely. The RTOS disables all interrupts during critical sections of RTOS operation causing uncertainty in the interrupt latencies. As the operating system needs to have total control on the executable firmware, the needs of Efore Powernet Oy's firmware development are not met.

2.3 FreeRTOS

FreeRTOS is selected for review to represent a pre-emptive/co-operative, considerably complex and free RTOS.

FreeRTOS is developed and maintained by Real Time Engineers Ltd. FreeRTOS's market share in 2017 was 20% (AspenCore 2017).

2.3.1 Review against Efore Powernet Oy's needs

FreeRTOS does not define the peripheral drivers, so the microcontroller manufacturer's drivers can be used.

FreeRTOS acknowledges the risks originating from dynamic memory allocation. Since ver. 9.0.0 FreeRTOS has included an option to allocate all kernel objects statically at compile time (Barry 2016, p. 25).

FreeRTOS does not enforce any additional operating system control over the interrupt handling. Operating system calls are required only if the interrupt needs to interact with

tasks. There are crtitcal RTOS sections, where interrupts are disabled. In similar fashion as in the case of uC/OS III, latency is introduced. FreeRTOS offers a way to disable only certain interrupts during critical sections of code (Barry 2016, pp. 182–233).

FreeRTOS has the possibility to delay the tasks by the count of system ticks. A built in macro can be used to convert a delay from milliseconds to system ticks (Barry 2016, pp. 65–75).

FreeRTOS doesn't allow any sections of the code to be executed outside the control of the RTOS. All of the functionality of the firmware must be divided into tasks (Barry 2016, pp. 46–47).

2.3.2 Conclusion

All of the application code must be controlled by the FreeRTOS and be included in the tasks handled by the operating system. Therefore, the requirements set for RTOS, defined in section 2.1.1 are not fulfilled.

2.4 Salvo

Third selection to the review is Salvo, developed and maintained by Pumpkin Inc.

Salvo was chosen to represent a co-operative, lightweight and proprietary RTOS.

2.4.1 Review against Efore Powernet Oy's needs

In similar fashion as the previous reviewed operating systems, Salvo does not define the peripheral drivers, so the microcontroller manufacturer's drivers can be used.

Salvo doesn't require interrupts to signal the RTOS. The RTOS has critical sections, where

interrupts are disabled, yet with pro version it is possible to disable only certain interrupts during critical sections of RTOS code (Kalman 2010, pp. 230–231).

Salvo also allocates all memory statically at compile time (Kalman 2010, pp. 87–88, 101–102, 108,110,114–115, 117).

Salvo allows delaying tasks with the resolution of the system tick counter (Kalman 2010, p. 198).

Salvo allows code to be run outside of the RTOS control, but it is discouraged. As the tasks are infinite loops, to step out from the RTOS requires all tasks to go into blocked state, which makes the code execution outside of the RTOS undeterministic (Kalman 2010, p. 232).

2.4.2 Conclusion

The undeterministic code execution does not meet the requirements of Efore Powernet Oy's firmware development.

2.5 Summary of the review of the selected RTOSes

All of the reviewed RTOSes allow the usage of peripheral drivers preferred by the user. FreeRTOS and Salvo fulfil the memory allocation, interrupt handling, and latency requirements. Each RTOS also has tools for making user programmable delays for the tasks.

The review shows, that none of the reviewed RTOSes suits the requirements set for the firmware development in Efore Powernet Oy. The main reason is that the µC/OS-III and FreeRTOS don't allow the execution of user code outside the RTOS control. Salvo allows code execution outside of the RTOS control, but that is too undeterministic to fill the requirements.

3 Designing the scheduler in C language

This chapter describes the design and implementation of a custom RTOS, which is able to fulfil the requirements specified in 2.1.1. As the specifications are based on task control and no other RTOS elements are required, the design considers only the scheduler of an RTOS.

The design is implemented in C language and the implementation is device independent as Efore Powernet Oy uses microcontrollers from various manufacturers. Device independent design defines that the implementation is independent from peripheral control registers, for instance.

The implementation is done using Keil µVision integrated development environment (IDE) (ARM 2017) and Efore Powernet Oy's ADC8580 battery charger as the test hardware unit (Powernet 2017).

3.1 Design concept

The design work starts from the conceptual level of what the scheduler must be able to do. Most of the specifications are defined as limitations or boundaries for the design. The delay specification is the main requirement.

The timing concept can be thought as a coin stack, where the coins represent tasks. The stack has two sized coins, small ones and a big one. The big coin represents the timing task, where the small coins are tasks that the scheduler executes. The idiomatic stack of coins is placed upon a surface with a hole in it, which represents the execution of the tasks. The hole is big enough for the small coins to fall through, but the big coin doesn't fit. Fig. 4 illustrates the concept.

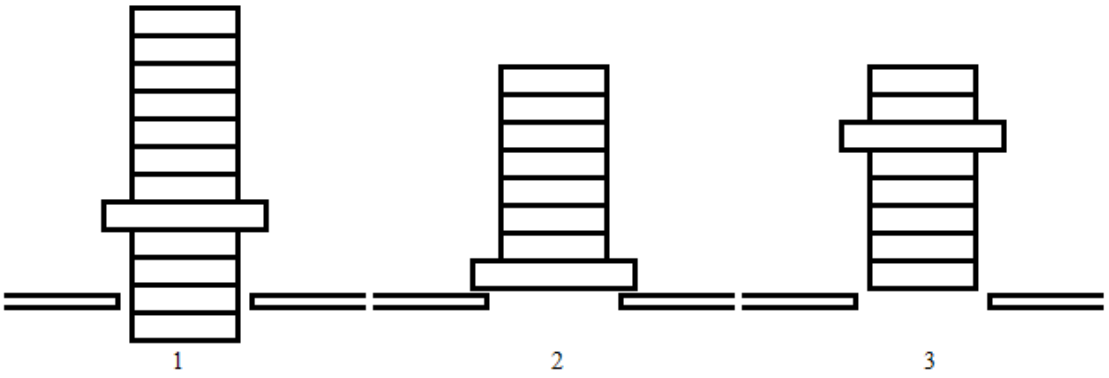

Fig 4. Illustration of the task timing concept “a coin stack”. In (1) the small coins (tasks) are able to fall freely through the hole (executed), until in (2) the larger coin (the timing object) blocks the hole. In (3) the larger coin is moved back a few positions (system tick) and thus the smaller coins are free to fall through the hole again.

All the task objects have time information, when they are supposed to be executed. The time information is a multiple of system ticks. The system tick is provided by a 1 ms timing instance of the microcontroller. When the system tick occurs, the timing object gets moved back for one tick’s time. This is done by the scheduler by reading the time information from the task objects above the timing task in the “coin stack” until the time value exceeds the current tick count. This allows the execution of tasks below the new slot of the timing object. This means that the tasks above the timing object are blocked and the tasks below are to be executed before "the big coin blocks the hole" again.

3.2 Scheduler core

The scheduler core is built as a linked list and it contains the tools for list modification. The core also contains the means to execute the tasks.

3.2.1 Structures and lists

The task objects are constructed as structures. A structure in C language is a construct, that can contain one or more variables of different data types (Kernighan & Dennis 1978, pp. 119–121). To form a queue of the task objects, they need to be connected together using pointers. A structure can contain a pointer to a structure with the same type (Kernighan & Dennis 1978, pp. 130–134).

Structures pointing to the next structure form a linked list. Adding a pointer also to the previous structure forms a two way linked list. Structures in linked lists are called nodes. Fig. 5 shows an example of a two way linked list with three nodes.

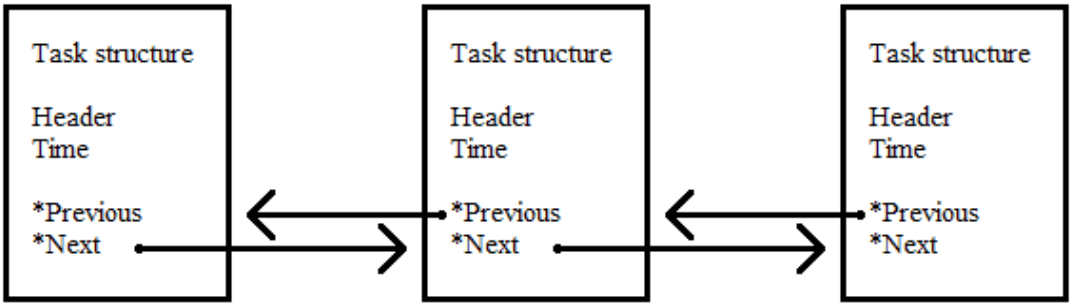

Fig 5. Two way linked list. The boxes represent the nodes and the arrows represent the pointers.

The task object structure contains unique identification information, called the header. It also contains the scheduled execution time of the task. And most importantly the task object contains pointers to next and previous task object structures. Following is the definition of the task object structure:

```
struct pn_queueobj {
  pn_queueobj *P_Previous;  /* link to previous object on the queue */
  pn_queueobj *P_Next;      /* link to next object on the queue */
  uint16_t Header;          /* unique identification */
  uint32_t Time;            /* execution time */
};
```

Adding a node to a list requires redirecting the pointers, as shown in Fig 6. A Node is added to the list, when a task is requested for scheduling.

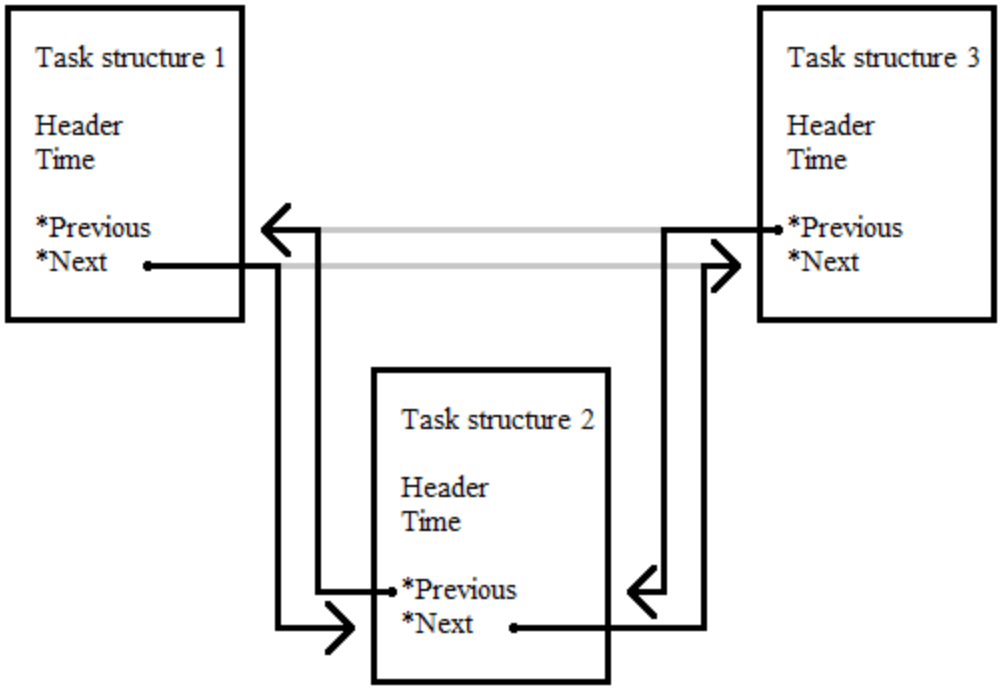

Fig 6. Adding a node to a list. The '*Next' pointer of node 1 and the '*Previous' pointer of node 3 are redirected to point to the added node 2. The '*Previous' pointer of node 2 is directed to point to node 1 and the '*Next' pointer of node 2 is directed to point to node 3.

Removing a node from a list also requires redirecting the pointers, as shown in Fig 7. A node is removed, when it's task gets executed or the scheduling of the task is cancelled.

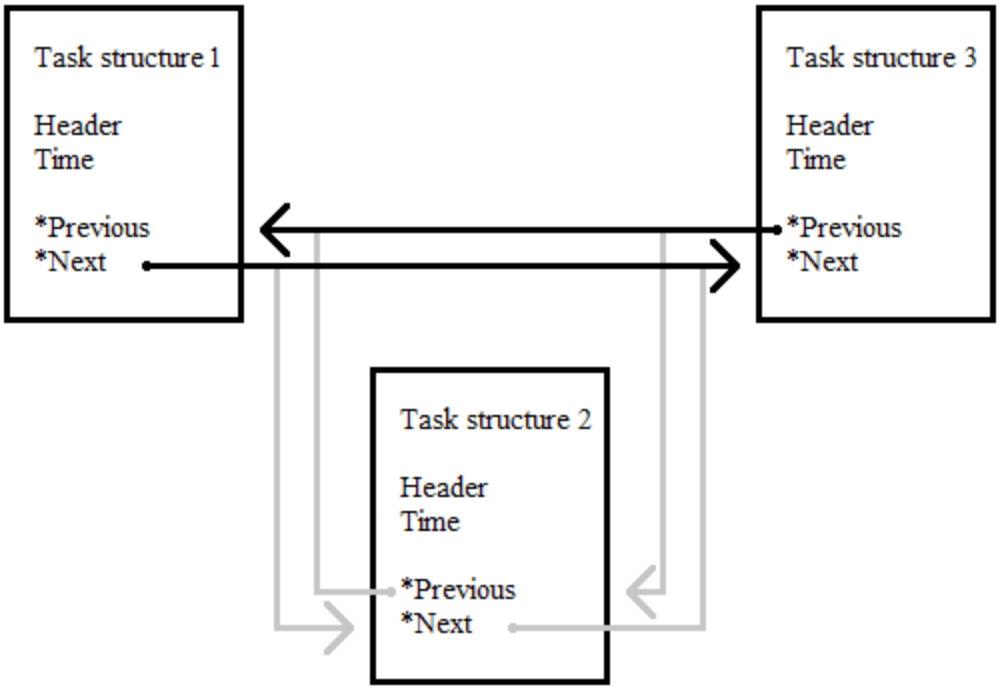

Fig 7. Removing a node from a list. The '*Next' pointer of node 1 is redirected to point to node 3 and the '*Previous' pointer of node 3 is redirected to point to node 1. The '*Next' and '*Previous' pointers of node 2 are reset.

3.2.2 Task object types and scheduling

The header of the task object gives the task a unique identifier but it also defines the type of the task. The design uses two types of tasks: timed and untimed. Timed tasks are low priority background tasks that are executed either periodically or delayed after an event. Untimed tasks are of high priority which are always executed before the timed ones. Untimed tasks can be for example peripheral driver related data processing, such as a serial port interrupt for received data, which triggers the need for data processing.

There are also three special objects on the list: the head, the tail and the timing object. The head is the first and the tail is the last object on the list. The timing object must be located somewhere between the head and the tail. Fig 8. shows an example of a list of tasks with all the mandatory objects and some untimed and timed tasks. Henceforth the list forms a queue to be executed.

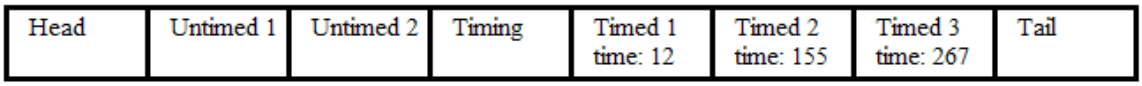

Fig 8. An example queue with the head in the beginning, the tail in the end and the timing object amongst the tasks.

To execute a task, the scheduler analyses the object which is pointed by the head. for the timing object, the scheduler does nothing and releases the execution to the code that is outside of the control of the operating system. For the task object the scheduler removes the object from the queue and executes the task's function call.

The scheduler code running on the main loop is as follows:

```
pn_queueobj *LP_HandledObject = 0;
LP_HandledObject = GP_QueueHead->P_Next;
if (!(LP_HandledObject->Header == _QUEUE_HEADER_QHEAD ||
      LP_HandledObject->Header == _QUEUE_HEADER_QTAIL)) {
  P_QueueHandler(LP_HandledObject, 0, _QUEUE_REMOVE);
  if (LP_HandledObject->Header & _QUEUE_HEADER_TIMED_MASK) {
    G_NowRunningFRC = LP_HandledObject->Time;
  }
  switch (LP_HandledObject->Header) {
    case _QUEUE_HEADER_TIME:
      break;
    default:
      if (LP_HandledObject->Header != _QUEUE_HEADER_TIME) {
        Core(LP_HandledObject);
      }
      break;
  }
}
```

First a temporary pointer (*LP_HandledObject) is introduced and it is assigned to point to the object next to the list head. Then it's checked that it isn't the head or the tail. The object is removed from the queue by calling the P_QueueHandler function with the argument

_QUEUE_REMOVE. The function P_QueueHandler can be found from Appendix A and the function related definitions are listed in Appendix B. The timing object cannot be removed, so no check for that needs to be done. Next the header is checked for the type of the task – timed or untimed. If it is a timed task, the global variable for the current operating system time (G_NowRunningFRC) gets updated. This operating system time is a free running counter (FRC), which counts the system ticks and is used to set delays in the scheduling function. Last step is the context switch to the task code. This is done in the Core function described below in an example with two tasks – a timed housekeep and an untimed communication:

```
void Core(pn_queueobj *LP_HandledObject) {
  switch (LP_HandledObject->Header) {
    case _QUEUE_HEADER_TIMED_HOUSEKEEP:
      HousekeepHandler();
      break;
    case _QUEUE_HEADER_COMMUNICATION:
      CommunicationHandler();
      break;
    default:
      break;
  }
}
```

After the task has been executed, the scheduler isn't called until the next turn of the main loop. This allows the execution of code which is outside the operating system's control after each task and always when the timing object is in front of the queue.

The timing object gets updated when a corresponding timer or other source of tick triggers. This event raises a flag (a binary variable) that is checked outside the scheduler. When the flag is true, the timing object gets rescheduled by 1 tick as presented in the following example:

```
if (G_Tick_Time)
{
  G_Tick_Time = 0;
  P_QueueHandler(&G_Obj_T_Time, 1, _QUEUE_SCHEDULE);
}
```

3.2.3 Initialization

Before the operating system is able to start scheduling, the tasks must be initialized. The requirement for memory allocation dictates that the structures associated with the operating system must be allocated statically during compilation. The initialization function must set the task object data to the structures as shown in the examples in Appendix C.

3.3 Usage during run time

After the scheduler has been initialized and the task structures and queue created, the scheduler is used during runtime.

3.3.1 Timed task scheduling example

The first example of scheduler usage is a timed housekeep task of a battery charger. The housekeeping task is scheduled to run periodically and it is used to translate ADC result from binary data to volts and amperes. The task also checks for over current at the charger output and polls and measures digital input signal from user interaction. The task is first scheduled after the initialization and it schedules itself before context switch.

The following listing shows an example of the code with undeclared variables and functions. ‘HousekeepHandler’ is the function which the housekeep task calls when the task gets executed:

```
void HousekeepHandler(void) {
  uint8_t L_i = 0;
  sint16_t L_temp = 0;

  /* Convert raw ADC values to real physical quantities */
  for (L_i = 0; L_i < _NUMBER_OF_CONVERSIONS; L_i++) {
    L_temp = ConvertADC(G_RawValues[L_i], G_Factors[L_i]);
    G_ConvertedValues[L_i] =  L_temp;
  }

  /* Monitor and handle possible over current */
  if (G_ConvertedValues[_OUTPUT_CURRENT] > _OUTPUT_CURRENT_LIMIT) {
    HandleOutputOverCurrent();
  }

  /* Read the digital input and react */
  if (DigitalInput()) { /* Input is active */
    DigitalInputActive();
  }
  else { /* Input is released */
    DigitalInputReleased();
  }

  /* Re-schedule the housekeep task */
  P_QueueHandler(&G_Obj_T_Housekeep, 1, _QUEUE_SCHEDULE);
}
```

3.3.2 Untimed task scheduling example

The second usage example is an untimed communication handler task. It is scheduled to run by the communication peripheral hardware driver event, when the communication frame receiving is finished. The incoming data is then processed and a reply frame prepared for transmitting with the driver.

The following listing shows an example of the code with undeclared variables and functions. ‘CommunicationHandler’ is the name of the function that is called, when the

communication task gets executed from the queue:

```
void HardwareDriver(void) {
  /* ... */
  if (FrameRecieved()) {
    P_QueueHandler(&G_Obj_Communication, 0, _QUEUE_SCHEDULE);
  }
  /* ... */
}

void CommunicationHandler(void) {
  ParseCommunicationFrame();
  PrepareReply();
  SendReply();
}
```

In this example the ‘P_QueueHandler’ is called with delay 0. This task cannot be delayed as the header defines it as untimed task and thus the scheduler will ignore any delay given as a parameter.

3.3.3 Task removing example

The third usage example is handling a timeout event. In a case where a required action is unable to react within a specified time frame, the failure may need a handling task. In this example the firmware triggers an action and sets a timeout for action reset. If the reaction is received in the given time frame, the timeout task is removed from scheduling.

The following listing shows an example of the code with undeclared variables and functions.

```
void Action(void) {
  /* ... */
  P_QueueHandler(&G_Obj_T_Timeout, 100, _QUEUE_SCHEDULE);
}


void ReactionReceived(void) {
  P_QueueHandler(&G_Obj_T_Timeout, 0, _QUEUE_REMOVE);
  /* ... */
}

void TimeoutHandler(void) {
  ResetAction();
  /* ... */
}
```

In this example the firmware triggers an event and calls function ‘Action’. This function triggers a timeout task with delay of 100. If the required reaction doesn’t occur in time, the task function ‘TimeoutHandler’ is called. If the reaction occurs before the timeout limit, the function ‘ReactionReceived’ is called which removes the timeout task from the scheduler queue by calling ‘P_QueueHandler’ with argument ‘_QUEUE_REMOVE’.

3.4 Scheduler performance analysis

The performance analysis is made against a simple round-robin type embedded design as a comparison study. The comparison is focused on the performance overhead against the number of tasks. Any initializations are not taken into account as the performance is analysed in the main loop during runtime. To calculate only the overhead, all the tasks are considered to be blocked to get the idle run of the main loop estimated.

The overhead is divided into static overhead that is constant regardless of the number of the tasks, and dynamic overhead that is defined by the number of tasks. The overheads are calculated by necessary statements, comparisons and function calls.

3.4.1 Simple round-robin

Simple round-robin is an infinite loop checking which tasks to run with trivial if-statements.

Following is a code example of a round-robin task control design with undeclared functions and variables:

```
while (1) {
  if (ConditionForA == 1) {
    ConditionForA = 0;
    HandleA();
  }
  if (ConditionForB == 1) {
    ConditionForB = 0;
    HandleB();
  }
  if (ConditionForC == 1) {
    ConditionForC = 0;
    HandleC();
  }
}
```

The example shows that the number of comparisons and if-statements grow linearly with the number of tasks in the design. With three tasks there are three if-statements, and comparisons in the loop. All of the overhead is dynamic.

3.4.2 Calculating scheduler overhead

The scheduler code in the main loop was discussed in section 3.2.2, from which the number of required if-statements, comparisons and function calls can be calculated. As all tasks are in blocked state, the queue contains only the timing object. This gives three if-statements as the switch/case structure can be considered as an if-statement, three

comparisons and a function call. The function call must be followed to count the provided statements. The function is shown in Appendix A and it contains two if-statements and four comparisons. Therefore, the scheduler overall overhead is five statements, seven comparisons and one function call.

All of the overhead in the scheduler is static. The number of tasks does not affect the idle loop time.

3.4.3 Comparison estimates

To calculate the time required to run the comparisons, if-statements and functions, some estimates must be made. The number of actual machine language instructions vary based on the microcontroller, but the following estimates can be justified, as the comparison isn't made between different microcontrollers and affects both designs.

A comparison in C language is divided into smaller parts in machine language. First the comparable variables are loaded into registers – two instructions – and then the actual comparison – one instruction. The comparison takes three instructions.

An if-statement uses the result of a comparison which is used to generate a suitable branch command – one instruction in machine language.

A function call in machine language requires saving the state of the current execution – stack pointer, program counter and the working registers, estimating 10 instructions. Then the passing of arguments and the jump command – estimate five instructions. Last when returning from the function call, the state of the previous execution must be resumed – estimating 10 instructions. This totals to 25 instructions.

Table 1 summarizes the machine language instructions needed for comparison, if-statement and function call.

Table 1. Summary of machine language instruction counts for different C language expressions.

Expression	Number of machine language instructions
Comparison	3
If / Switch statement	1
Function call	25

3.4.4 Comparison

The round-robin instruction count sums up from three instructions per task for the comparison and one instruction per task for the if-statement. This totals to four instructions per task.

The scheduler instruction count from five if-statements gives 5 instructions, seven comparisons gives 21 instructions and one function call gives 25 instructions totalling to 51 instructions.

With a low number of tasks, the round-robin is more efficient, yet with bigger number of tasks the scheduler becomes more efficient. The cross over point with these estimates comes from dividing the scheduler estimate with the round-robin estimate (51 / 4 = 12.75). With 13 tasks or more the scheduler's overhead overcomes round-robin method overhead.

Table 2 summarizes the expression and instruction counts for the round-robin and the scheduler.

Table 2. Summary of machine language instruction counts for compared designs.

		Count	Total instructions
Round-robin (all dynamic)	Comparisons	1	3
	If-statements	1	1
	Functions	0	0
	Total		4
Scheduler (all static)	Comparisons	7	21
	If-statements	5	5
	Functions	1	25
	Total		51

4 Impact on the firmware design in product development

Efore Powernet Oy has used round-robin based firmware design before the designed custom scheduler discussed in Chapter 3. The analysis of the impact of the custom scheduler is done by comparing the design work against the previous design methods.

4.1 Focus of the impact analysis

To review different designs analytically, a group of focus points are selected for analysis:

- Number of tasks in the design
- Timing needs of the design
- Firmware structure
- Firmware design time
- Code readability and maintainability

The hardware initializations are left out of the focus as the usage of the scheduler does not affect those.

4.2 Analysis of designs without the scheduler

In the designs where the scheduler is not used, the functions in the main loop are not considered as tasks in the design work. The functions are used to clear the clutter from the main loop to make the code more readable.

The main loop follows flags set by interrupt handlers which require the data processing involved with the peripherals triggering the interrupts.

One of the flags is usually a timer interrupt to produce a system tick behaviour to count delays. Most used system tick time is 1 ms. This flag has the longest string of if-statements, as all the delays in the system are using the same tick. All of the delays require their own variables for counting.

Following is a simplified code example with undeclared functions and variables:

```
while (1) {
  if (HW_PeripheralA_Flag == 1) {
    HW_PeripheralA_Flag = 0;
    HandleHW_PeripheralA();
  }
  if (HW_PeripheralB_Flag == 1) {
    HW_PeripheralB_Flag = 0;
    HandleHW_PeripheralB();
  }
  if (Tick1ms == 1) {
    Tick1ms = 0;
    if (CtrTaskA++ == 100) {
      CtrTaskA = 0;
      TaskA();
    }
    if (CtrTaskB++ == 200) {
      CtrTaskB = 0;
      TaskB();
    }
  }
}
```

This example shows that the structure of the code gets more complex when the number of tasks or functions with delays grows larger. When the delays are conditional with other functions, the structure becomes hard to read and implementation maintainability is more difficult without introducing possible problems elsewhere.

4.3 Examples of projects using the scheduler

For this review two different firmware projects are selected. The selection is made to review the scheduler applicability in two different use cases.

The first reviewed project is a train auxiliary power system's controller unit. This unit controls up to four power converters through controller area network (CAN) and handles communication with the train vehicle controller through another CAN bus. It handles the load sharing between converters and disables excessive converters when the power demand is lower (Saxell 2019, pp. 16–18).

The second reviewed project is a battery charger with battery state of health monitoring (Powernet 2017). The charger measures the battery impedance for condition monitoring and analysis. The charger uses Modbus for monitoring and maintenance purposes (Modbus 2012).

4.4 Impact on train power supply system controller design

The train power supply system controller unit (SCU) does not use round-robin scheduling due to concerns of the high complexity and maintainability of the firmware. This project is the driving force for the scheduler development shown in this thesis.

This project has 63 tasks handled by the scheduler. It also has 19 tasks outside the scheduler.

39 of the scheduler handled tasks are timed tasks and the scheduler handles all the timing needs of the project.

The firmware structure is layered with hardware drivers in the lowest layer, scheduler in the highest layer and application tasks in between.

As the number of the tasks is higher than the critical limit of 13 discussed in section 3.4.4, the scheduler significantly improves the performance of the design compared to round-

robin.

4.5 Impact on battery charger design

The scheduler is used in this project to evaluate its impact on the design of the battery charger and whether the performance fulfils the hard real time requirements of the power supply control and monitoring.

This project has 16 tasks handled by the scheduler. It also has seven tasks outside the scheduler. The tasks outside the scheduler are Modbus driver related function calls. The timing requirement of those tasks are such that the selected system tick of 1 ms is too slow for the Modbus application.

15 of the 16 scheduler handled tasks are timed tasks so the scheduler handles all the timing needs of the project except the Modbus.

The firmware structure is layered with hardware drivers in the lowest layer, scheduler in the highest layer and application tasks in between.

As the number of the tasks is higher than the critical limit of 13 discussed in section 3.4.4, the scheduler improves the performance of the design compared to round-robin.

The scheduler can handle the hard real time requirements of the charger. The most critical requirements are to be able to measure load current reliably and to switch off the load with an internal semiconductor switch if the load current goes above specification for longer than 50 ms.

5 Conclusions and discussion

The market review of existing RTOSes concluded, that the selected RTOSes could not completely cover all the set requirements. The selection of the reviewed RTOSes covers different kinds of RTOSes to see how they would compare against the requirements.

The review revealed one common incompatibility with Efore Powernet Oy's needs. None of the reviewed RTOSes could run code reliably outside the control of the RTOS. This incompatibility can thus be seen as a very common feature in readily available RTOSes on the market. As a conclusion can be said that there are no RTOSes on the market to fill the requirements of Efore Powernet Oy's firmware design needs.

The lack of readily available RTOSes set a need to design a custom scheduler. The design was based on the requirements set for the market review and thus the scheduler complies with all of the requirements.

In an embedded system design, the operating system cannot compromise the performance of the main application under any operating conditions. The performance of the scheduler was compared against a simple round-robin design and as a conclusion, the results showed the scheduler to overcome the round-robin after 13 tasks. Therefore in basically any real life power supply control case, the use of the presented scheduler is justified, as the number of tasks is higher than 13. In addition the code structuring and further maintainability improves with the use of the proposed scheduler.

The assessment of the impact on designs using the scheduler, revealed the impact to the Efore Powernet Oy's firmware development to be positive. The usage of the scheduler has improved the maintainability and readability of the code as the clutter of the round-robin delay handling with separate counters and complexity is not present.

In overall this study shows that Efore Powernet Oy needed a custom scheduler to be designed and the resulting scheduler has had a positive impact on the firmware design.

For future work the scheduler performance could be improved by looking into the overhead described in section 3.4.4. The function call overhead could be decreased by restructuring the queue handling function into two separate functions: add and remove. The

remove function could be then inlined to remove the necessity to perform an actual function call at that point.

Also as an improvement for the code in the main loop, the handled object should be compared to the timing object before calling the queue handler with ‘remove’ argument. This removes the necessity for the function call in the idle loop and decreases overhead.

The design of the scheduler can be studied further by reviewing the need and possibility to implement more RTOS features to accompany the scheduler and to upgrade the design to be a complete RTOS.

References

ARM 2017. Getting started with MDK. [online document]. [Referred on 20.10.2019]. Available on http://www2.keil.com/docs/default-source/default-document-library/mdk5-getting-started.pdf?sfvrsn=2[NC,L]

AspenCore 2017. 2017 Embedded Markets Study. [online document]. [Referred on 20.10.2019]. Available on https://m.eet.com/media/1246048/2017-embedded-market-study.pdf

Barry, R. 2016. Mastering the FreeRTOS Real Time Kernel [online document]. [Referred on 20.10.2019]. Available on https://www.freertos.org/wp-content/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

Heath, S. 2003. Embedded Systems Design. Oxford: Elsevier

Kalman, A. 2010. Salvo User Manual [online document]. [Referred on 20.10.2019]. Available on http://www.pumpkininc.com/content/doc/manual/SalvoUserManual.pdf

Kernighan, B. & Dennis, M. 1978. The C Programming Language. London: Prentice-Hall

Labrosse, J. 2011. µC/OS-III The Real-Time Kernel For the STM32 ARM Cortex-M3. Weston: Micriμm Press

LaVerne, D. 1989. C in Embedded Systems and the Microcontroller World. National Semiconductor Application Note 587 [online document]. [Referred on 20.10.2019]. Available on http://cpu-ns32k.net/files/AN587.pdf

Modbus 2012. Modbus application protocol specification v1.1b3. [online document]. [Referred on 20.10.2019]. Available on http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

Mundra, T. S. & Kumar A. 2007. An Innovative Battery Charger for Safe Charging of NiMH/NiCd Batteries. *IEEE Transactions on Consumer Electronics*, vol. 53, issue 3, pp. 1044–1052.

Nemati, F. 2012. Resource sharing in real-time systems on multiprocessors. Västerås: Mälardalen University

Powernet 2017. ADC8000 series. [online document]. [Referred on 20.10.2019]. Available on https://powernet.fi/wp-content/uploads/2018/10/ADC8000-DS-29032017.pdf

Saxell, T. 2019. Train auxiliary power supply system's specification. [online document]. [Referred on 20.10.2019]. Available on http://urn.fi/URN:NBN:fi-fe2019090526756

Appendix A

The code for the function P_QueueHandler.

#include <pn_queue_handler.h>

```
#include <pn_queue_objects.h>

/* Globals */
pn_queueobj *GP_QueueHead;
pn_queueobj *GP_QueueTail;
uint32_t G_NowRunningFRC;
uint32_t G_FRC;

/* P_QueueHandler
 * Purpose:
 * Handles queue modifications.
 * Inputs:
 * pn_queueobj *LP_HandledObject:
 * Queue object to be modified
 * uint32_t L_Delay:
 * Time offset for next execution (ms, for timed/1ms objects only)
 * uint8_t L_Cmd:
 * Command to perform on the object
 * _QUEUE_SCHEDULE (0): (re)schedule object
 * _QUEUE_REMOVE (1): remove object from queue
 * Outputs:
 * -
 * Globals read:
 * G_NowRunningFRC:
 * the base time to apply the L_Delay offset
 * Globals modified:
 * the queue (objects, head and tail)
 */
```

```
void P_QueueHandler(pn_queueobj *LP_HandledObject,
                    uint32_t L_Delay, uint8_t L_Cmd) {
  pn_queueobj *LP_Temp = 0;
  uint16_t L_RetValue = 0;
  switch (L_Cmd)
  {
    case _QUEUE_SCHEDULE:
      /* Re-link next and previous if rescheduling */
      if (LP_HandledObject->P_Previous != 0 &&
          LP_HandledObject->P_Next != 0) {
        LP_HandledObject->P_Previous->P_Next = LP_HandledObject->P_Next;
        LP_HandledObject->P_Next->P_Previous = LP_HandledObject->P_Previous;
        LP_HandledObject->P_Previous = 0;
        LP_HandledObject->P_Next = 0;
      }
      /* Timed object */
      if (LP_HandledObject->Header & _QUEUE_HEADER_TIMED_MASK)
      {
        /* First check if the object can be placed at the end of the queue */
        if(!(GP_QueueTail->P_Previous->Header & _QUEUE_HEADER_TIMED_MASK) ||
            (L_Delay >= (GP_QueueTail->P_Previous->Time - G_NowRunningFRC))) {
          LP_HandledObject->P_Previous = GP_QueueTail->P_Previous;
          LP_HandledObject->P_Next = GP_QueueTail;
          GP_QueueTail->P_Previous->P_Next = LP_HandledObject;
          GP_QueueTail->P_Previous = LP_HandledObject;
          LP_HandledObject->Time = G_NowRunningFRC + L_Delay;
        }
        /* Search the spot from the queue */
        else
        {
          LP_Temp = GP_QueueHead->P_Next;
          while (!(LP_Temp->Header &  _QUEUE_HEADER_TIMED_MASK) &&
                   (LP_Temp->Header != _QUEUE_HEADER_QTAIL)) {
            LP_Temp = LP_Temp->P_Next;
          }
          while ((L_Delay >= (LP_Temp->Time - G_NowRunningFRC)) &&
                  (LP_Temp->Header != _QUEUE_HEADER_QTAIL)) {
            LP_Temp = LP_Temp->P_Next;
          }
```

```
    LP_HandledObject->P_Next = LP_Temp;
    LP_HandledObject->P_Previous = LP_Temp->P_Previous;
    LP_Temp->P_Previous->P_Next = LP_HandledObject;
    LP_Temp->P_Previous = LP_HandledObject;
    LP_HandledObject->Time = G_NowRunningFRC + L_Delay;
  }
}
/* Untimed object */
else {
  /* Check if the head of the queue is timed and place the object at the
     head */
  if (GP_QueueHead->P_Next->Header & _QUEUE_HEADER_TIMED_MASK) {
    LP_HandledObject->P_Next = GP_QueueHead->P_Next;
    LP_HandledObject->P_Previous = GP_QueueHead;
    GP_QueueHead->P_Next->P_Previous = LP_HandledObject;
    GP_QueueHead->P_Next = LP_HandledObject;
  }
  /* Place the object in front of the timed ones */
  else {
    LP_Temp = GP_QueueHead->P_Next;
    while (!(LP_Temp->Header &  _QUEUE_HEADER_TIMED_MASK) &&
            (LP_Temp->Header != _QUEUE_HEADER_QTAIL)) {
      LP_Temp = LP_Temp->P_Next;
    }
    LP_HandledObject->P_Next = LP_Temp;
    LP_HandledObject->P_Previous = LP_Temp->P_Previous;
    LP_Temp->P_Previous->P_Next = LP_HandledObject;
    LP_Temp->P_Previous = LP_HandledObject;
  }
}
break;
```

```
    case _QUEUE_REMOVE:
      /* 1ms object is never removed, only rescheduled */
      /* Head and tail are never removed or rescheduled */
      if (LP_HandledObject->Header != _QUEUE_HEADER_TIME &&
          LP_HandledObject->Header != _QUEUE_HEADER_QHEAD &&
          LP_HandledObject->Header != _QUEUE_HEADER_QTAIL) {
        /* Check if the object is in the queue… */
        if ((LP_HandledObject->P_Next != 0) &&
            (LP_HandledObject->P_Previous != 0)) {
          /* ...and remove it */
          LP_HandledObject->P_Previous->P_Next = LP_HandledObject->P_Next;
          LP_HandledObject->P_Next->P_Previous = LP_HandledObject->P_Previous;
          LP_HandledObject->P_Next = 0;
          LP_HandledObject->P_Previous = 0;
        }
      }
      break;
    default:
      /* Should not be reached */
      break;
  }
}
```

Appendix B

Header file “pn_queue_handler.h”

```
/* QueueHandler bit masks */
#define _QUEUE_HEADER_TIMED_MASK  0x8000

/* QueueHandler Commands */
#define _QUEUE_SCHEDULE           0
#define _QUEUE_REMOVE             1

/* Type definitions */
typedef struct pn_queueobj pn_queueobj;
struct pn_queueobj {
  pn_queueobj *P_Previous;  /* link to previous object on the queue */
  pn_queueobj *P_Next;      /* link to next object on the queue */
  uint16_t Header;          /* unique identification */
  uint32_t Time;            /* execution time */
};

/* Global variables */
extern uint32_t G_NowRunningFRC;
extern uint32_t G_FRC;

extern pn_queueobj *GP_QueueHead;
extern pn_queueobj *GP_QueueTail;

/* Function prototypes */
extern void P_QueueHandler(pn_queueobj*, uint32_t, uint8_t);
extern void Core(pn_queueobj*);
```

Appendix C

Examples of files “pn_queue_objects.h” and “pn_queue_objects.c”

pn_queue_objects.h:

```
/* Function prototypes */
void InitQueueObjects(void);
void Init_Q_Object (pn_queueobj* LP_Object, uint16_t ObjHeader);

/* Objects and their headers, OS objects first */
#define _QUEUE_HEADER_QHEAD             0x0000
extern pn_queueobj G_Obj_QHead;

#define _QUEUE_HEADER_QTAIL             0x7FFF
extern pn_queueobj G_Obj_QTail;

#define _QUEUE_HEADER_TIME              0xFFFE
extern pn_queueobj G_Obj_T_Time;

#define _QUEUE_HEADER_TIMED_HOUSEKEEP   0xFFFD
extern pn_queueobj G_Obj_T_Housekeep;

#define _QUEUE_HEADER_COMMUNICATION     0x0002
extern pn_queueobj G_Obj_Communication;
```

pn_queue_objects.c:

```
#include <pn_queue_handler.h>
#include <pn_queue_objects.h>

/* Introduce object variables, OS objects first */
pn_queueobj G_Obj_QHead;
pn_queueobj G_Obj_QTail;
pn_queueobj G_Obj_T_Time;

pn_queueobj G_Obj_T_Housekeep;
pn_queueobj G_Obj_Communication;

/* Function to initialize single objects */
void Init_Q_Object (pn_queueobj* LP_Object, uint16_t ObjHeader) {
  LP_Object -> P_Previous = 0;
  LP_Object -> P_Next = 0;
  LP_Object -> P_Params = 0;
  LP_Object -> Header = ObjHeader;
  LP_Object -> Time = 0;
  LP_Object -> Elevated = 0;
}

/* This init function must be called at the startup before the main loop */
void InitQueueObjects(void) {
  /* Initialize OS objects and form the basic queue with head-timing-tail */
  GP_QueueHead = &G_Obj_QHead;
  GP_QueueTail = &G_Obj_Qtail;

  G_Obj_QHead.P_Previous = 0;
  G_Obj_QHead.P_Next = &G_Obj_T_Time;
  G_Obj_QHead.P_Params = 0;
  G_Obj_QHead.Header = _QUEUE_HEADER_QHEAD;
  G_Obj_QHead.Time = 0;
  G_Obj_QHead.Elevated = 0;
```

```
  G_Obj_QTail.P_Previous = &G_Obj_T_Time;
  G_Obj_QTail.P_Next = 0;
  G_Obj_QTail.P_Params = 0;
  G_Obj_QTail.Header = _QUEUE_HEADER_QTAIL;
  G_Obj_QTail.Time = 0;
  G_Obj_QTail.Elevated = 0;

  G_Obj_T_Time.P_Previous = GP_QueueHead;
  G_Obj_T_Time.P_Next = GP_QueueTail;
  G_Obj_T_Time.P_Params = 0;
  G_Obj_T_Time.Header = _QUEUE_HEADER_TIME;
  G_Obj_T_Time.Time = 1;
  G_Obj_T_Time.Elevated = 0;

  /* Initialize task objects */
  Init_Q_Object(&G_Obj_T_Housekeep, _QUEUE_HEADER_TIMED_HOUSEKEEP);
  Init_Q_Object(&G_Obj_Communication, _QUEUE_HEADER_COMMUNICATION);
}
```